Helion

Wydajność i optymalizacja kodu

Istota dynamiki działania oprogramowania

Richard L. Sites

Przedmowa Luiz André Barroso, Google Fellow

Tytuł oryginału: Understanding Software Dynamics

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-9515-2

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/roprwy
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

Rozdział 1.

Mój program działa zbyt wolno

Ktoś wchodzi do mojego gabinetu i mówi: „Mój program działa zbyt wolno". Po chwili przerwy pytam: „A jak wolno powinien działać?".

Dobry programista ma gotową odpowiedź na takie pytanie, ponieważ potrafi opisać zadanie do wykonania i oszacować, ile czasu powinna zająć każda jego część. Taka osoba może odpowiedzieć tak: „To zapytanie do bazy danych sprawdza 10 000 rekordów, z których mniej więcej 1000 okazuje się potrzebnych. Każdy dostęp powinien zajmować około 10 milisekund, a używanych jest 20 dysków, tak więc 10 000 operacji dostępu powinno w sumie zająć mniej więcej 5 sekund. Sieć nie jest obciążona, a procesor i pamięć są używane w niewielkich i prostych operacjach znacznie szybszych niż dostęp do dysku. Całe zapytanie zajmuje około 15 sekund, co jest zbyt długim czasem".

Mniej staranny programista może odpowiedzieć tak: „Pracowałem całą noc i napisałem 1000 wierszy kodu z użyciem wielu istniejących bibliotek. Kod działa, ale zapytanie zajmuje 15 sekund, a chciałem uzyskać poziom $^1/_{10}$ sekundy. Jedna z bibliotek musi być zbyt wolna. Jak mogę ją wykryć?". Taki programista nie ma pojęcia, czy zasadne jest oczekiwanie czasu na poziomie $^1/_{10}$ sekundy. Nie wie też, ile czasu powinno zajmować każde wywołanie kodu z biblioteki lub czy poprawnie korzysta z bibliotek. Nie ma też zaprojektowanego mechanizmu obserwowania dynamiki działania kodu, aby ustalić, na co przeznaczany jest czas programu. W tej książce omawiam wszystkie te zagadnienia.

1.1. Kontekst centrum danych

W tym miejscu wprowadzam pojęcia i koncepcje ze złożonego środowiska oprogramowania. Twoje środowisko może być znacznie prostsze, ale opisywane kwestie można prawie dokładnie do niego przenieść. Używana tu terminologia pochodzi ze świata centrów danych, jednak te same zagadnienia występują również w kontekście baz danych, komputerów stacjonarnych, pojazdów, gier, a także w innych środowiskach z ograniczeniami czasowymi.

Transakcja, *zapytanie* lub *żądanie* to komunikat wejściowy dla systemu komputerowego przetwarzany jako odrębna jednostka pracy. Każdy komputer przetwarzający transakcje nazywam *serwerem*. *Latencja* lub *czas odpowiedzi* dla transakcji to czas między jej wysłaniem a otrzymaniem wyniku. *Oferowane obciążenie* to liczba transakcji przesyłanych na sekundę. Jeśli ta wartość przekracza liczbę transakcji przetwarzanych na sekundę, następuje wydłużenie czasu reakcji (czasem bardzo znaczne). *Usługa* to kolekcja programów obsługujących transakcje jednego rodzaju. Duże centra danych przetwarzają jednocześnie transakcje dziesiątek różnych usług, a każda usługa ma inne oferowane obciążenie i inne cele związane z latencją.

Latencja transakcji nie jest stała. Ma rozkład prawdopodobieństwa oparty na tysiącach transakcji wykonywanych na sekundę. *Latencje z długiego ogona* dotyczą najwolniejszych transakcji z rozkładu. Prostym sposobem na zestawienie takich transakcji jest ustalenie latencji dla percentyla 99%, czyli czasu przekraczanego przez najwolniejsze 1% wszystkich transakcji. Jeśli oferowane obciążenie wynosi 5000 transakcji na sekundę, ten 1% to 50 transakcji.

Dynamika działania programu lub ich grupy oznacza ich aktywność w czasie — jakie fragmenty kodu są wykonywane w różnych momentach, na co oczekują, ile pamięci zajmują i jak poszczególne programy wpływają na siebie nawzajem. Programiści wyobrażają sobie prostą dynamikę działania programu, jednak w rzeczywistości program może, od czasu do czasu, zachowywać się zupełnie inaczej i pracować znacznie wolniej, niż tego oczekujemy. Jeśli uda się zaobserwować prawdziwą dynamikę, można będzie dostosować obraz w umyśle. Zwykle pozwala to na poprawę wydajności kodu za pomocą prostych zmian.

Przedmiotem zainteresowania są tu transakcje z udziałem użytkownika realizowane w złożonym oprogramowaniu, na przykład w centrum danych do obsługi telefonów komórkowych. Ważne są transakcje, które zwykle są realizowane szybko, ale czasami wymagają znacznie więcej czasu, na tyle, że użytkownik końcowy odczuwa irytujące opóźnienie. W centrach danych budżet na sprzęt dla każdej usługi jest często zależny od tego, ile transakcji na sekundę każdy serwer może obsługiwać. Ta docelowa liczba jest określana *empirycznie*, przez zwiększanie oferowanego obciążenia do momentu przekroczenia jakiegoś progowego ograniczenia czasowego dla latencji z długiego ogona w rozkładzie. Następnie ustalane jest nieco niższe obciążenie docelowe.

Jeśli uda Ci się zrozumieć sytuację, a następnie zmniejszyć liczbę zbyt długich transakcji, ten sam sprzęt będzie mógł obsłużyć wyższe obciążenie i nadal realizować cele związane z latencją z długiego ogona. Nie będzie to wymagało ponoszenia dodatkowych kosztów. Takie rozwiązanie zapewnia duże korzyści finansowe. Uzdolniony i mający trochę szczęścia inżynier wydajności może czasem wprowadzić w oprogramowaniu prostą zmianę, która da oszczędności warte jego 10-letniego wynagrodzenia. Firmy i klienci cenią takich ludzi.

Oprogramowanie, w którym transakcje mają ograniczenia czasowe, znacznie różni się od oprogramowania wsadowego lub działającego w trybie offline (a także od większości oprogramowania testowego). Ważną miarą w oprogramowaniu z transakcjami jest czas odpowiedzi, natomiast w oprogramowaniu wsadowym istotną miarą jest zazwyczaj wydajność wykorzystania sprzętu. W transakcjach ważny jest nie tyle *średni* czas odpowiedzi, co najdłuższe czasy, czyli latencja z długiego ogona.

W centrach danych preferowana jest zwykle wyższa średnia latencja z niższą latencją z długiego ogona niż odwrotna sytuacja. Większość osób dojeżdżających do pracy ma podobne preferencje i woli o kilka minut dłuższą drogę, która zawsze zajmuje tyle samo czasu, niż krótszą, na której występują godzinne opóźnienia.

W oprogramowaniu wsadowym średnie obciążenie procesora na poziomie 98% może być korzystne. W oprogramowaniu transakcyjnym 98% obciążenia to *katastrofa*, a nawet średnie obciążenie na poziomie 50% może być zbyt wysokim poziomem, ponieważ skutkuje długimi czasami odpowiedzi, gdy oferowane obciążenie rośnie skokowo na kilka sekund do poziomu trzykrotnie przekraczającego średnie. Gdy w 2004 roku trafiłem do firmy Google, procesory w centrum danych średnio były zajęte przez 9% i bezczynne przez 91% czasu. Obciążenie na poziomie 9% było zbyt niskie. Zwiększenie go do 18% bez wydłużania latencji z długiego ogona pozwoliło podwoić efektywność wszystkich centrów danych. Ponowne podwojenie poziomu, do 36%, byłoby korzystne, jednak kolejne podwojenie, do 72%, prawdopodobnie naruszyłoby zbyt wiele ograniczeń czasowych transakcji.

W trakcie analizowania wydajności złożonego oprogramowania transakcyjnego zakładam w tej książce, że badane programy działają poprawnie i ich średnia szybkość pracy jest wystarczająco wysoka. Nie omawiam tu projektowania lub debugowania oprogramowania. Nie staram się też zrozumieć lub zwiększyć średniej wydajności. Ponadto przyjmuję, że transakcje, które zawsze działają powoli, zostały zidentyfikowane i poprawione w środowiskach testowych lub diagnostycznych w trybie offline, gdzie nie ma ograniczeń czasowych. Dlatego do zbadania pozostały już tylko okazjonalnie powolne transakcje. Skupiam się na mechanizmach, które powodują długi czas pracy takich transakcji, na tym, jak zaobserwować te mechanizmy, a także na tym, jak zinterpretować poczynione obserwacje.

Gdy używasz telefonu komórkowego do wysyłania wiadomości tekstowych, czytania wpisów, przeszukiwania internetu, przeglądania mapy, strumieniowania filmu, używania aplikacji, a nawet wybierania numeru telefonu, gdzieś działa centrum danych, które reaguje na Twoje żądania. Jeśli odpowiedzi są irytująco powolne, a jakaś konkurencyjna aplikacja lub usługa działa szybciej, może to spowodować, że przejdziesz na nowe rozwiązanie, a przynajmniej zaczniesz rzadziej korzystać z powolnego produktu. W ekosystemie rozwiązań z ograniczeniami czasowymi każdy ma motywację (często finansową) do tego, by ograniczyć irytujące opóźnienia. Jednak niewiele osób potrafi to robić.

Ta książka ma sprawić, że takich osób będzie więcej.

1.2. Sprzęt w centrach danych

W dużych centrach danych w budynku znajduje się mniej więcej 10 000 serwerów. Każdy taki serwer to komputer o wielkości zbliżonej do komputera stacjonarnego, ale bez obudowy. Około 50 płyt serwerów jest zamontowanych w szafce, a mniej więcej 200 szafek znajduje się w bardzo dużym pomieszczeniu. Typowy serwer ma od 1 do 4 gniazd na procesory, z których każdy ma po 4 – 50 rdzeni, masę[1] pamięci RAM, kilka dysków twardych lub nośników *SSD*

[1] To pojęcie z dziedziny informatyki oznaczające 10^{12}.

i połączenie z siecią centrum danych działającą w topologii switching fabric, co umożliwia każdemu serwerowi komunikację ze wszystkimi pozostałymi. Ponadto przynajmniej niektóre serwery mogą komunikować się z internetem, a tym samym z Twoim telefonem. Poza budynkiem znajdują się duże generatory, które zapewniają energię całemu budynkowi (w tym klimatyzacji) na kilka dni lub tygodni na wypadek awarii zasilania. Wewnątrz umieszczone są akumulatory, które potrafią podtrzymywać pracę serwerów i przełączników sieciowych przez kilkadziesiąt sekund do czasu uruchomienia generatorów.

Na każdym serwerze działa wiele programów. Zwykle nie ma uzasadnienia biznesowego przydzielanie niektórych serwerów tylko do obsługi poczty elektronicznej, innych tylko do kafelek map, a jeszcze innych do obsługi komunikatorów. Zamiast tego na każdym serwerze pracuje wiele programów, a każdy z nich zwykle ma wiele *wątków*. Na przykład program serwera poczty elektronicznej może mieć 100 wątków roboczych przetwarzających jednocześnie żądania od tysięcy użytkowników, z których większość pisze lub czyta wiadomości. Wiele tych wątków może być aktywnych i oczekiwać na dostęp do dysku lub inne warstwy oprogramowania. Wątki robocze przyjmują przychodzące żądania, realizują je, odpowiadają, a następnie przechodzą do następnego oczekującego żądania od innego użytkownika. W godzinach z największym obciążeniem prawie wszystkie wątki robocze są zajęte. W godzinach, gdy aktywność użytkowników jest mniejsza, przynajmniej połowa wątków jest bezczynna i oczekuje na zadania. W prawie wszystkich skalach czasowych (mikrosekund, milisekund, sekund, minut) występuje ciągły cykl wzrostów i spadków poziomu oferowanego obciążenia. Istnieje nawet siedmiodniowy cykl z niższą aktywnością w soboty i niedziele (zgodnie z dniami roboczymi w zachodnim świecie).

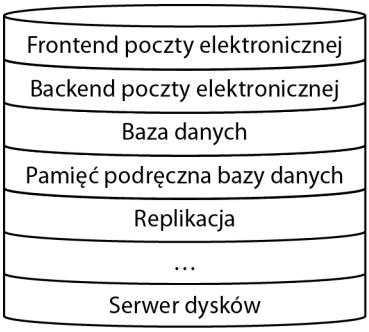
Aby móc kontrolować czas odpowiedzi, ważne jest mieć wolne zasoby sprzętowe dostępne dla transakcji z udziałem użytkowników, ponieważ obciążenie ze strony użytkowników okresowo skokowo rośnie z powodu wydarzeń w świecie fizycznym. Jednak ekonomiczne jest też wykonywanie niekomunikujących się z użytkownikami programów wsadowych, uruchamianych, gdy procesory są bezczynne. Obok komunikujących się z użytkownikami programów pierwszego planu i uruchamianych w tle programów wsadowych na każdym serwerze zawsze działa kilka programów nadzorujących, które śledzą, jak obciążony jest serwer, ile błędów na nim występuje, ile wolnej pamięci dyskowej pozostało itd. Te programy nadzorujące odpowiadają za stan maszyn oraz ich ponowne uruchamianie i rekonfigurowanie przez włączanie, zatrzymywanie i restartowanie różnych programów. Okazuje się, że takie środowisko jest skomplikowane już na jednym serwerze. Pomnóż teraz tę złożoność razy 10 000 serwerów w pomieszczeniu.

1.3. Oprogramowanie w centrum danych

Oprogramowanie dla centrów danych różni się od samodzielnych programów i oprogramowania testowego. Składa się z *wielu warstw* podsystemów, z których liczne działają równolegle i obsługują żądania od wielu różnych usług i licznych instancji pojedynczych usług. Każdy podsystem próbuje zwrócić odpowiedź wystarczająco szybko, aby osiągnąć stawiane mu cele z obszaru latencji. Często się zdarza, że warstwy obsługujące jedno żądanie użytkownika działają na różnych serwerach. Aby zwiększyć wydajność, w wielu warstwach używana jest programowa pamięć podręczna do przechowywania niedawno potrzebnych danych

lub obliczonych wyników. Znalezienie i ponowne użycie danych zapisanych w pamięci podręcznej nazywane jest *trafieniem*; z kolei brak danych w pamięci podręcznej to *chybienie*. Dalej zobaczysz, że dynamika działania pamięci podręcznych może w zaskakujący sposób wpływać na latencję transakcji.

Gdy użytkownik żąda tekstu e-maila, żądanie najpierw trzeba skierować do centrum danych zawierającego główne repozytorium poczty elektronicznej tego użytkownika, a następnie przekazać do serwera równoważącego obciążenie, który to serwer przekieruje je do którejś z mniej obciążonych maszyn spośród setek serwerów odpowiedzialnych za frontend poczty elektronicznej. Warstwa frontendu zarządza żądaniem i ostatecznie generuje wynik w formacie HTML lub zgodny z API aplikacji. Ta warstwa żąda danego e-maila od warstwy backendu, która kieruje wywołanie do *warstwy bazy danych*, ta wywołuje warstwę pamięci podręcznej bazy danych, a jeśli nastąpi chybienie, kieruje żądanie do warstwy replikacji (aby uzyskać dostęp do rezerwowego repozytorium poczty elektronicznej z innego centrum danych). Ostatecznie zgłaszane jest wywołanie do warstwy serwera dysków, który wczytuje e-mail z jednego z kilku redundantnych dysków, co ilustruje rysunek 1.1. Następnie wyniki są zwracane przez całe drzewo wywołań i często są przy tym modyfikowane.

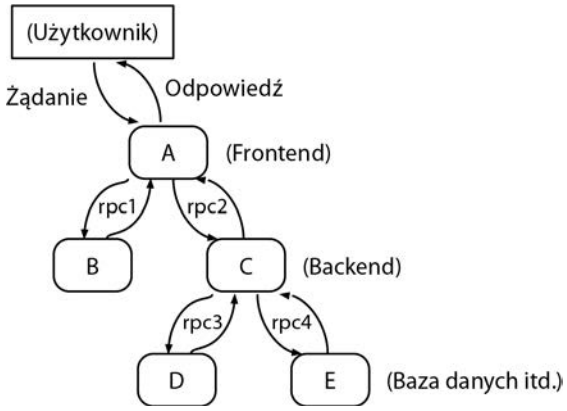


Rysunek 1.1. *Warstwy przykładowego oprogramowania*

Wszystkie te operacje są łączone za pomocą jakiejś formy przekazywania komunikatów lub zdalnych wywołań procedur (ang. *remote procedure call* — RPC). W tej książce używam skrótu „RPC”. Wywołania RPC z jednej warstwy do drugiej mogą być przetwarzane synchronicznie (jednostka wywołująca oczekuje wtedy na odpowiedź) lub asynchronicznie (jednostka wywołująca kontynuuje pracę i może zgłaszać inne wywołania RPC wykonywane równolegle na wielu różnych serwerach). To właśnie równoległe wykonywanie niewielkich porcji pracy umożliwia oprogramowaniu w centrum danych wykonywanie dla jednego żądania olbrzymich ilości zadań i kończenie ich w ułamku sekundy. Jedna transakcja z udziałem użytkownika może skutkować użyciem od 200 do 2000 różnych serwerów.

Na rysunku 1.2 pokazane jest niewielkie przykładowe drzewo wywołań RPC w stylu z pracy [Sigelman 2010]. Serwer A kieruje synchroniczne wywołanie do serwera B, a po zwróceniu sterowania przez B serwera A kieruje wywołanie do serwera C. Serwer C może równolegle wywołać serwery D i E, a następnie oczekiwać na zwrócenie przez nie odpowiedzi.

Każde żądanie od użytkownika i każde podżądanie RPC mają cel w postaci określonego czasu odpowiedzi. Jeśli dla żądania od użytkownika do warstwy frontendu poczty elektronicznej celem jest 200 milisekund, dla warstwy backendu cel może wynosić 160 milisekund, dla warstwy



Rysunek 1.2. *Drzewo wywołań RPC dla pięciu serwerów (A – E)*

bazy danych może być jeszcze krótszy itd., aż do 50 milisekund dla serwera dysków. Gdy podżądanie zbyt długo generuje odpowiedź, każda jednostka wywołująca też jest narażona na przekroczenie czasu. Dla zestawu równoległych wywołań używane jest pojęcie *asymetria czasów wykonania* (ang. *execution skew*) określające zmienność czasu wykonywania zadań.

Jeśli równolegle przetwarzanych jest wiele wywołań RPC, zwykle to najwolniejsze z nich określa łączny czas odpowiedzi. Dlatego jeżeli wykonujesz równolegle 100 wywołań RPC, to łączny czas odpowiedzi jest zależny od czasu na poziomie percentyla 99%. Asymetria czasów wykonania sprawia, że ważne jest zrozumienie i kontrolowanie długich czasów odpowiedzi.

1.4. Latencja z długiego ogona rozkładu

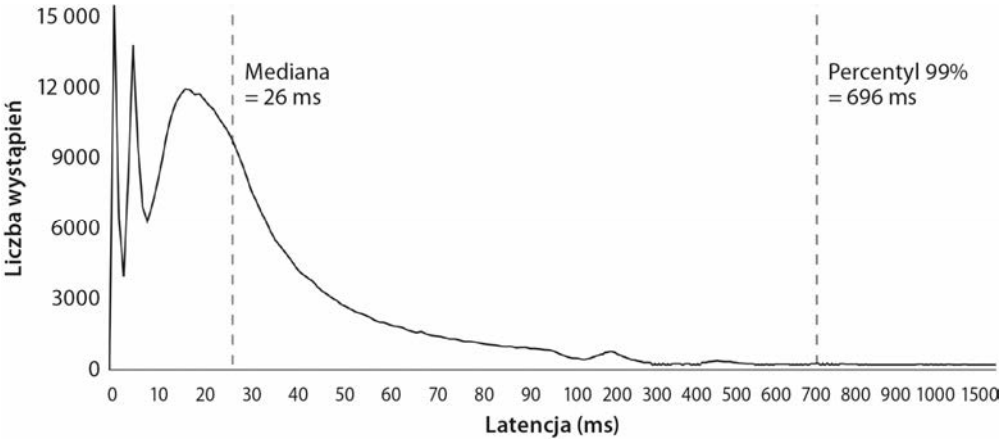
Latencja to czas zegarowy, jaki upływa między dwoma zdarzeniami. W trakcie precyzyjnego omawiania latencji trzeba określić, jakie są to zdarzenia. Na przykład „latencja wywołania RPC" może oznaczać czas między wysłaniem żądania przez program w trybie użytkownika (czyli klienta w terminologii modelu klient-serwer) a otrzymaniem przez niego odpowiedzi na to żądanie. Może też oznaczać czas od otrzymania żądania przez wywołany program w trybie użytkownika (serwer w terminologii modelu klient-serwer) do momentu wysłania odpowiedzi. Latencja w tych dwóch różnych definicjach (dla klienta i dla serwera) dla tego samego wywołania RPC może się różnić o 30 milisekund lub nawet więcej, co może skłaniać do zastanowienia się nad tym, co się stało z dodatkowym czasem i w których urządzeniach komputerowych lub sieciowych on zaginął.

W tej książce skupiam się na latencji wywołań RPC na *serwerze*, chyba że omawiam rozbieżności takie jak 30 milisekund w tym przykładzie.

Żądania RPC kierowane do usługi mają różną latencję, jednak często dla podobnych żądań latencja grupuje się wokół podobnych wartości. Można to zilustrować na histogramie wartości latencji, czyli na wykresie przedstawiającym na osi X przedziały latencji, a na osi Y liczbę wywołań RPC z poszczególnych przedziałów.

W transakcjach w centrum danych histogramy latencji mają jeden lub kilka wierzchołków dla standardowych przypadków i często *długi ogon* znacznie dłuższych czasów odpowiedzi dla nietypowych przypadków [Blake 2015, Hoff 2012, Weaveworks 2017, Dean 2013].

Na histogramie dotyczącym serwera dysków (rysunek 1.3) występują trzy wierzchołki z czasem około 1 milisekundy, 3 milisekund i 20 milisekund dla trzech różnych rodzajów standardowych przypadków, a następnie długi ogon, który ciągnie się do ponad 1500 milisekund. Pożądany czas odpowiedzi wynosi 50 milisekund lub mniej. Ta książka ma pomóc w zrozumieniu i skróceniu długiego ogona. Słabo zaznaczone wierzchołki na poziomie tuż przed 250 milisekundami, 500 milisekundami, 750 milisekundami itd. wskazują na źródło tej zagadki związanej z wydajnością. Jej rozwiązanie znajdziesz w części II.



Rysunek 1.3. *Histogram obrazujący dostęp do serwera dysków; po prawej stronie widoczny jest długi ogon*

Histogram latencji warto przedstawić za pomocą tylko kilku liczb, zamiast używać 500 wartości, jeśli wyznaczonych jest 500 przedziałów. Jakie wartości należy wybrać?

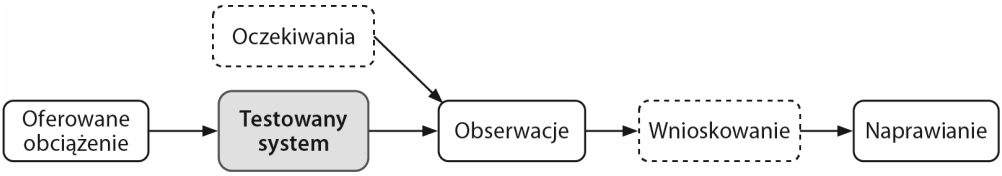
Mediana (lub podobna średnia) latencji jest wyjątkowo słabym opisem asymetrycznego lub wielowierzchołkowego rozkładu, ponieważ rzadko wypada w pobliżu dużej liczby rzeczywistych wartości i nic nie mówi o kształcie ani wielkości długiego ogona, który jest tu interesujący. Na wykresie z rysunku 1.3 mediana wynosi 26 milisekund, co nie niesie żadnych informacji na temat wierzchołków lub ogona. Także maksymalna latencja jest kiepskim wyborem, ponieważ w ciągu dnia może wystąpić jedno skrajnie długie wywołanie RPC (wynikające na przykład z odzyskiwania pamięci lub sprzętowego błędu dysku), podczas gdy wszystkie inne są dziesiątki lub nawet setki razy krótsze.

Zamiast tego lepiej posłużyć się percentylami. Jeśli histogram latencji reprezentuje 50 000 pomiarów, najkrótszych 500 to najszybszy 1%, a najdłuższych 500 to najwolniejszy 1%. Wartość percentyla 99% to liczbowa granica między najszybszymi 99% a najwolniejszym 1%; 99% posortowanych pomiarów jest mniejszych lub równych tej wartości. Istnieje wiele takich wartości. Wszystkie one wypadają między 49 500. a 49 001. posortowanym pomiarem. Można posłużyć się dowolną z tych liczb, jednak zwyczajowo stosuje się 49 500. wartość. Szybkim, ale przydatnym sposobem opisania rozkładu z długim ogonem jest podanie wartości percentyla 99%, 95%, 99,9% itd. Wartość percentyla 99% w histogramie na rysunku 1.3 wynosi 696 milisekund. Jest ona zbyt wysoka w porównaniu z celem równym 50 milisekund. Oznacza to poważny problem z wydajnością.

Z rozdziału 9. dowiesz się, co spowodowało ten konkretny długi ogon i jak problem został rozwiązany. Poznasz też nową wartość percentyla 99%, która wyniosła około 150 milisekund. Ta prosta zmiana dała zyski równe mojemu 10-letniemu wynagrodzeniu.

1.5. Model myślenia

W procesie analizy latencji z długiego ogona rozkładu i powiązanych problemów z wydajnością stosuję następujące podejście: najpierw należy oszacować, ile czasu jakieś zadanie powinno zająć, następnie trzeba zaobserwować, ile czasu rzeczywiście zajmuje, a na końcu zastanowić się nad różnicami między tymi wartościami. Rysunek 1.4 przedstawia ten model.



Rysunek 1.4. *Model analizowania wydajności złożonego oprogramowania*

Ten model ujmuje testowany system obejmujący oprogramowanie i sprzęt, oferowane obciążenie tego systemu, oczekiwania co do wydajności systemu przy danym obciążeniu, obserwacje dynamiki działania i wydajności za pomocą narzędzi, wnioskowanie przez człowieka na temat tego, co się dzieje, i ostatecznie wprowadzenie poprawek lub zmian w celu zwiększenia wydajności.

1.6. Szacowanie rzędu wielkości

W trakcie badania wydajności oprogramowania jednym z aspektów pytania „Jak wolne powinno być?" jest oszacowanie, ile czasu powinny zajmować określone zadania. Nawet gdy te szacunki są bardzo ogólne, mogą zapewnić cenne informacje. Programiści zwracający uwagę na wydajność nieustannie szacują w głowach rzędy wielkości, gdy projektują i piszą ważne programy.

Określenie *rząd wielkości* dotyczy przybliżonej oceny wielkości liczby. Rząd wielkości w systemie *dziesiętnym* wymaga przedstawienia szacunków za pomocą potęg liczby 10 (1, 10, 100 itd.). Gdy określany jest rząd wielkości w systemie *dwójkowym*, należy podać szacunki za pomocą potęg liczby 2 (1, 2, 4, 8 itd.). Możesz się też zetknąć z pośrednimi dziesiętnymi rzędami wielkości — 1, 3, 10, 30 itd. W tej książce używam rzędów wielkości w systemie dziesiętnym, chyba że napiszę, że jest inaczej. Stosuję notację *O(n)*, co oznacza „rzędu *n*", i zawsze podaję *jednostki*. Jest bardzo ważne, czy chodzi o O(10) nanosekund, O(10) milisekund czy O(10) bajtów. Od tego miejsca używam zapisu *ns*, *μs* i *ms* jako skrótów jednostek czasu (nanosekund, mikrosekund i milisekund).

W tabeli 1.1 znajdziesz listę szacunków, które każdy poważny programista powinien znać. Pochodzi ona z wystąpienia Jeffa Deana (jednego z bardzo nielicznych posiadaczy tytułu Google Fellow) z 2009 roku. Od tamtego czasu podane wartości dużo się nie zmieniły. Dodałem tylko kolumnę z rzędami wielkości.

Tabela 1.1. *Liczby, które każdy powinien znać [Dean 2009]*

Operacja	Czas	O(n)
Dostęp do pamięci podręcznej L1	0,5 ns	O(1) ns
Błędna predykcja gałęzi	5 ns	O(10) ns
Dostęp do pamięci podręcznej L2	7 ns	O(10) ns
Zajęcie lub zwolnienie muteksu	25 ns	O(10) ns
Dostęp do pamięci głównej	100 ns	O(100) ns
Kompresja 1 KB za pomocą biblioteki Zippy	3000 ns	O(1) µs
Przesyłanie 2 KB przez sieci o szybkości 1 Gb/s	20 000 ns	O(10) µs
Sekwencyjny odczyt 1 MB danych z pamięci	250 000 ns	O(100) µs
Przesyłanie pakietu tam i z powrotem w tym samym centrum danych	500 000 ns	O(1) ms
Wyszukiwanie danych na dysku	10 000 000 ns	O(10) ms
Sekwencyjny odczyt 1 MB danych z dysku	20 000 000 ns	O(10) ms
Przesyłanie pakietu na trasie Kalifornia->Holandia->Kalifornia	150 000 000 ns	O(100) ms

Oszacowanie rzędu wielkości oczekiwanego czasu wykonywania różnych części programu sprawia, że po uzyskaniu rzeczywistych pomiarów tych czasów można łatwo dostrzec miejsca, w których wyniki znacznie różnią się od oczekiwań. **Właśnie to umożliwia naukę**. Czasem szacunki są błędne i dowiesz się czegoś nowego na temat działania komputerów lub programów. Czasem szacunki będą poprawne, ale okaże się, że program działa niezgodnie z Twoimi wyobrażeniami, a nieoczekiwane szybsze lub wolniejsze operacje wymagają poprawek. Gdy nabierzesz wprawy i Twoje szacunki staną się trafniejsze, coraz więcej znalezionych rozbieżności będzie oznaczać rzeczywiste błędy z obszaru wydajności.

Znajomość szacunków z tabeli 1.1 pomoże Ci też identyfikować prawdopodobne źródła błędów związanych z wydajnością. Jeśli wykonywanie jakiegoś fragmentu programu zajmuje 100 ms więcej, niż oczekujesz, problem prawdopodobnie nie jest związany z błędną predykcją gałęzi, ponieważ wpływ tego aspektu jest 10 000 000 mniejszy niż 100 ms. Bardziej prawdopodobnym źródłem problemu są operacje dyskowe lub sieciowe albo, o czym przekonasz się w dalszych rozdziałach, długi czas zajmowania blokad lub oczekiwania na długie podżądania RPC.

Dalej zaprojektuję i zbuduję narzędzia do obserwacji i wyświetlania danych. Gdy będziesz z nich korzystać, postaraj się wyrobić sobie nawyk prognozowania (w formie rzędu wielkości), czego spodziewasz się zobaczyć. Po nabraniu wprawy w realizowaniu cyklu predykcje--obserwacje-porównania błyskawicznie zaczniesz dostrzegać nietypowe wyniki.

1.7. Dlaczego transakcje działają powoli?

Warto przypomnieć, że szczególnie interesujące są transakcje, które zwykle działają szybko, ale w niektórych sytuacjach transakcje wymagają dużo dodatkowego czasu — na tyle dużo, że użytkownik końcowy odczuwa irytujące opóźnienie. Powolne transakcje to przede wszystkim te,

które naruszają zapisany cel związany z czasem odpowiedzi. Co może powodować takie opóźnienia? Co może być przyczyną zmiennych latencji, a przede wszystkim długiego ogona w ich rozkładzie?

Wskazówką jest to, że dana transakcja (lub określony typ transakcji) jest zwykle szybka. Gdy jest powolna, mamy do czynienia ze standardowym czasem wykonywania zwiększonym o jakieś nieznane opóźnienie. Jeśli uda się zidentyfikować źródło tego opóźnienia, zwykle można wprowadzić proste zmiany w kodzie, które eliminują większość tego opóźnienia, i w ten sposób skrócić długi ogon w rozkładzie latencji.

W oprogramowaniu z wieloma warstwami najczęstszym źródłem opóźnień w jednej warstwie jest oczekiwanie na odpowiedź z niższej warstwy. Najniższa wolna warstwa może działać długo z powodu własnej charakterystyki lub przeciążenia *przesadnym oferowanym obciążeniem*. W trakcie wyznaczania celów dotyczących czasu reakcji koniecznie pamiętaj o tym, aby jednocześnie określić cele (a precyzyjniej — *ograniczenia*) związane z oferowanym obciążeniem.

Zmiana kodu w jednej warstwie nie pomoże, jeśli czeka ona na niższą warstwę. Trzeba znaleźć najniższą warstwę, która jest zbyt wolna, i starać się poprawić jej działanie. Dlatego warto projektować oprogramowanie w taki sposób, aby umożliwić obserwowanie czasu działania każdej warstwy i przekształcać uzyskane pomiary na postać pozwalającą szybko dostrzec, co jest wąskim gardłem. Prostym wykresem tego typu jest zestawienie rzeczywistego oferowanego obciążenia z oczekiwanym oferowanym obciążeniem i rzeczywistego czasu odpowiedzi z oczekiwanym w każdej warstwie lub w każdym interfejsie wywołań RPC.

Jeśli oferowane obciążenie w warstwie N jest akceptowalne i warstwa ta nie oczekuje zbyt długo na warstwę niższego poziomu (N + 1), ale i tak czas odpowiedzi warstwy N jest zbyt długi, istnieją wykonywane na jednym serwerze wywołania RPC do warstwy N, których latencja często jest normalna, ale czasem znacznie dłuższa. Warto wtedy dokładnie przyjrzeć się danemu serwerowi. Możliwe, że powolne wywołania RPC wykonują *dodatkową pracę*, która normalnie nie jest realizowana, albo przetwarzają standardowe zadania, ale *działają powoli*, wolniej niż zwykle.

Dodatkowa praca wynika ze struktury gałęzi kodu i przechowywanego stanu. Programy bardzo się różnią między sobą, jeśli chodzi o to, co może być powodem wykonywania dodatkowych zadań. Jednak zwykle realizują dodatkową pracę nawet wtedy, jeśli program działa na serwerze sam, bez innych aplikacji. Tego typu błędy dotyczące wydajności są stosunkowo łatwe do wykrycia. Wystarczy uruchomić kod w trybie offline w środowisku testowym i przekazać do niego sklonowane lub zarejestrowane żądania generowane w związku z rzeczywistym obciążeniem, a przy tym zastosować dodatkową instrumentację w celu znalezienia wzorców rozgałęzień prowadzących do problemu. Wykonywanie kodu na maszynach testowych pozwala zastosować standardowe narzędzia do pomiaru wydajności, które spowalniają przetwarzanie 2-krotnie, a nawet 20-krotnie lub jeszcze bardziej. W książce Brendana Gregga [Gregg 2021] opisanych jest wiele narzędzi do obserwacji odpowiednich do użytku w takim środowisku.

Bardziej ciekawym przypadkiem (i tematem tej książki) są wywołania RPC, które wykonują normalne zadania, ale wolniej niż zazwyczaj. Oznacza to, że coś *oddziałuje* na normalną pracę wywołania RPC na jednym serwerze i spowalnia wykonywanie kodu. Takie transakcje nazywam *hamowanymi*. Ich opóźnienie zwykle nie ujawnia się w testach w trybie offline, a tylko w obliczu rzeczywistego obciążenia generowanego przez użytkowników i często wyłącznie w godzinie podczas dnia, gdy aktywność jest największa. Celem jest wykrycie źródeł oddziaływań i ich wyeliminowanie lub przynajmniej zminimalizowanie ich wpływu. Niestety, w środowisku

produkcyjnym narzędzia do obserwacji, które spowalniają przetwarzanie 2-krotnie czy nawet o 10%, zanadto wydłużają czas odpowiedzi, by można je zastosować. W aktywnych centrach danych (lub w pojazdach, grach wieloosobowych itd.) potrzebne są techniki i narzędzia do obserwacji z kosztami dodatkowymi mniejszymi niż 1%. W branży jest bardzo mało tego typu produktów. Jeden z nich poznasz w części III tej książki.

Warto przypomnieć, że w centrum danych na każdym serwerze działa wiele programów, a każdy z nich może mieć wiele wątków. Oddziaływania na jednym serwerze muszą być spowodowane przez jakieś czynniki z tej maszyny (może to być przychodzący i wychodzący ruch sieciowy). Oddziaływania w takim środowisku prawie zawsze są wynikiem rywalizacji o współdzielony zasób.

1.8. Pięć podstawowych zasobów

Są tylko cztery zasoby sprzętowe współdzielone przez niepowiązane programy działające na jednym serwerze:

- procesor,
- pamięć,
- dysk twardy lub nośnik SSD,
- sieć.

Jeśli w programie działa kilka współpracujących ze sobą wątków, występuje też piąty podstawowy zasób:

- sekcja krytyczna w oprogramowaniu.

Sekcja krytyczna to fragment kodu używający współdzielonych danych w sposób, który może skutkować nieprawidłowym działaniem programu, jeśli ten fragment będzie wykonywany przez więcej niż jeden wątek naraz. Tego rodzaju sekcje kodu są chronione za pomocą blokad programowych, aby jednocześnie mogły być wykonywane tylko przez jeden wątek. Inne wątki muszą wtedy czekać na wejście do sekcji krytycznej.

Aby wykryć oddziaływania, trzeba najpierw zrozumieć normalne działanie kodu. Punktem wyjścia jest nauczenie się technik precyzyjnego pomiaru pięciu podstawowych zasobów. W pozostałych rozdziałach części I omawiam cztery zasoby sprzętowe, a blokady programowe opisuję dopiero w rozdziale 27., kiedy odpowiednie narzędzie do obserwacji będzie już gotowe. Jeśli powolne wywołanie RPC próbuje użyć jednego z pięciu wymienionych zasobów, a inny program lub wątek także korzysta z danego zasobu, wywołanie RPC będzie musiało na niego poczekać. Jest to główny mechanizm będący źródłem oddziaływań lub hamowania.

1.9. Podsumowanie

Ta książka ma pomóc w zrozumieniu dynamiki działania transakcji w oprogramowaniu centrów danych, baz danych, komputerów stacjonarnych, gier i dedykowanych kontrolerów, a przede wszystkim tych transakcji, których działanie zajmuje znacznie więcej czasu niż zazwyczaj.

Dobry programista potrafi oszacować z precyzją na poziomie rzędu wielkości, jak dużo czasu powinno zająć wykonywanie danego fragmentu kodu. Dzięki temu umie też wykryć i poprawić kod, który zawsze działa zbyt wolno. W tej książce zakładam, że kod, który zawsze jest wykonywany zbyt długo, został już poprawiony. Interesuje nas znacznie trudniejszy do zrozumienia kod, który tylko czasem działa zbyt powoli.

Na serwerze w centrum danych, który przetwarza tysiące transakcji na sekundę, niektóre z nich będą niekiedy wykonywane powoli, a następnie ponownie szybko. Na histogramie czasów transakcji widać długi ogon wolnych przebiegów, które mają nieproporcjonalny wpływ na łączny czas odpowiedzi dla użytkowników i nieproporcjonalnie zmniejszają ilość pracy, jaką dany serwer potrafi wykonać. Te powolne transakcje są wynikiem jakiegoś rodzaju oddziaływań, jednak w wielowarstwowym oprogramowaniu w centrum danych często trudno jest ustalić, która warstwa jest powolna w określonej transakcji. Dlatego trudno też jest stwierdzić, gdzie należy szukać oddziaływań.

Szacunki rzędu wielkości, takie jak pokazane w tabeli 1.1, mogą być wskazówką przy identyfikowaniu możliwych źródeł lub mechanizmów związanych z problemami z wydajnością, jednak przeważnie nie da się precyzyjnie ustalić powolnego fragmentu kodu. Aby to zrobić, trzeba zaprojektować odpowiednie narzędzia do obserwacji warstwowego oprogramowania i serwerów wykonujących wiele niepowiązanych programów, które mogą wpływać na swoje działanie.

Na ogólnym poziomie transakcja na pojedynczym serwerze albo działa normalnie, albo działa powoli, albo oczekuje na coś z tego serwera. Dwa ostatnie z tych zjawisk są skutkiem interakcji. Dalej omawiam mechanizmy związane z tymi zjawiskami i pokazuję, jak zaobserwować je *in situ*.

I to tyle. Aby rozwiązać problemy z okazjonalnie powolnymi transakcjami, wystarczy: (i) zidentyfikować, która warstwa kodu działa powoli, a następnie (ii) ustalić, co wpływa na jej działanie, a potem (iii) wprowadzić poprawki. Reszta książki ma nauczyć Cię, jak wykonywać te trzy proste kroki. Niestety dwa pierwsze z nich nie są łatwe.

- Skupiam się na zrozumieniu okazjonalnie powolnych transakcji RPC.
- Gdy równolegle wykonywanych jest 100 wywołań RPC, czas z poziomu percentyla 99% jest wyznacznikiem *ogólnego* czasu odpowiedzi.
- Oprogramowanie w centrach danych jest podatne na asymetryczny rozkład czasu wykonywania, w którym występuje długi ogon bardzo wolnych odpowiedzi.
- Spowolnione lub hamowane transakcje oznaczają, że coś oddziałuje na pracę wywołań RPC.
- Oddziaływania wynikają ze współdzielenia pięciu podstawowych zasobów.
- Oddziaływania są trudne do zaobserwowania *in situ*. Dalej zbuduję brakujące narzędzia do obserwacji.
- Koniecznie oszacuj rząd wielkości oczekiwanych czasów wykonywania, aby ułatwić sobie dostrzeżenie nieoczekiwanych wyników.

Kod eksperta: tu każdy bit ma znaczenie!

Mimo że dysponujemy coraz szybszym i wydajniejszym sprzętem, oprogramowaniu wciąż stawia się wyśrubowane wymagania: musi działać błyskawicznie i niezawodnie. Nieraz od jakości pracy systemu zależy bardzo dużo, nawet bezpieczeństwo i życie człowieka. Eliminacja źródłowych przyczyn problemów wydajnościowych jest jednak niezwykle trudnym zadaniem. Wymaga wiedzy o działaniu sprzętu, interakcjach między komponentami systemu, a także wysokiej klasy umiejętności programistycznych.

Ta książka jest przeznaczona dla programistów i osób zaawansowanych w nauce programowania. Dzięki niej nauczysz się uwzględniać w projektach narzędzia do obserwacji pracy kodu i analizować uzyskane dzięki nim dane o wydajności. Dowiesz się też, jak na podstawie takiego wnioskowania uzyskiwać znaczną poprawę w szybkości przetwarzania powolnych operacji. Prezentowane treści zilustrowano przykładami i ćwiczeniami w języku C lub C++ w systemie Linux. Materiał zawarty w książce to trzy główne zagadnienia: pomiary, obserwacje, a także wnioskowanie i nanoszenie poprawek w kodzie. Wartościową częścią książki jest omówienie procesu budowy niskokosztowego narzędzia do obserwacji KUtrace i jego zastosowania we wdrażaniu wyrafinowanych rozwiązań programistycznych. W ten sposób można u źródła usuwać przyczyny problemów z wydajnością kodu.

Najciekawsze zagadnienia:

- problemy sprzętowe: procesory, pamięci, dyski twarde, nośniki SSD i sieci
- korygowanie kodu wolno działającego programu
- przydatne mechanizmy monitorowania pracy kodu
- analiza danych dotyczących wydajności
- identyfikacja problemów, takich jak wykonywanie zbyt wielu instrukcji, powolne wykonywanie instrukcji, oczekiwanie na zasoby i blokady programowe

---

DR RICHARD L. SITES zajmuje się programowaniem od 1959 roku. Zdobył olbrzymią wiedzę o interakcjach sprzętu i oprogramowania. Rozwijał mikrokod architektury VAX, był jednym z architektów mikroprocesora DEC Alpha i wymyślił liczniki wydajności powszechnie stosowane w procesorach. Zajmował się niskokosztowym śledzeniem mikrokodu i oprogramowania w firmach: DEC, Adobe, Google i Tesla. Posiada 66 patentów i jest członkiem National Academy of Engineering.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-9515-2

9 788328 395152

Cena: 119,00 zł

Pearson
Addison-Wesley